Treebeard : An Optimizing Compiler for Tree Based ML Inference

ABSTRACT

Decision tree ensembles, are a commonly used machine learning model. Such models are generated when using machine learning techniques like gradient boosting and random forests and are used in a wide range of applications from recommendation systems to tabular data analysis. They are also deployed at scale in the enterprise.

Several libraries such as XGBoost, LightGBM and Sklearn expose algorithms for both training and inference with decision tree ensembles. These systems incorporate a limited set of optimizations usually targeted at specific hardware. Further, existing systems do not specialize the inference code to the model being used.

This paper presents Treebeard, an extensible compiler for decision tree models. Treebeard progressively lowers inference code to LLVM IR through multiple intermediate representations. By applying model specific optimizations at the higher levels, loop optimizations at the middle level and machine specific optimizations lower down, Treebeard can specialize inference code for each model on each supported hardware target. Treebeard performs several novel optimizations such as tree tiling, tree walk unrolling and tree walk interleaving to improve performance of decision tree ensemble inference.

We implement Treebeard using the MLIR compiler infrastructure and demonstrate the utility of Treebeard by evaluating it on a diverse set of tree ensemble models. We report that Treebeard optimizations improve average latency over a batch of inputs by 2.2X in comparison to an unoptimized baseline. Further, we find that Treebeard is significantly faster than other frameworks like XGBoost (3X on average) and Treelite (5X on average) in both single and multi-core settings.

1. INTRODUCTION

Intro tex here!

2. COMPILER OVERVIEW

Figure 1 shows the high level structure of Treebeard. The input to TreeBeard is a JSON serialized decision forest model, it supports popular frameworks like XGBoost and LightGBM and is extensible to other frameworks. Given an input model our compiler generates optimized inference code. Specifically it generates a callable batch inference function `predictForest` that given an array of inputs, outputs an array of model predictions.

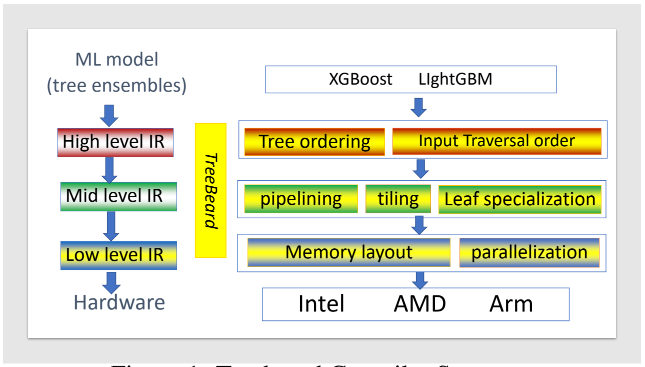

Figure 1: Treebeard Compiler Structure

TreeBeard specializes the code generated for inference by progressively optimizing and lowering a high level representation of the `predictForest` function down to LLVM IR [?]. It is implemented as a dialect in the MLIR framework [?], At this level (HIR) it applies transformations that determine how inputs traverse the decision forest. It re-order trees in the forest and determines the high level loop structure (process one tree at a time or one input at a time). It then lowers to a traditional loop based IR (MIR). Several loop level optimizations like unrolling and pipelining are applied at this level. At the lowest level of IR TreeBeard applies layout optimizations and generates parallel code in LLVM IR. LLVM is then used to JIT compile the generated IR to executable code. The following sections describe each level of the IR and their lowering in more detail.

TODO Should we describe the dialect's type system? TODO Kr : consider focusing on the example instead of verbose description of optimizations. This section can be short, descriptions can come in later sections.

2.1 High Level IR

Treebeard parses the input JSON file and generates a function with a single MLIR operation, `predictForest` that represents inference using the input model on a set of rows. The operation contains within it a tree based representation of the model that can be manipulated by optimizing transformations. Transformations on the model such as tiling, tree reordering and leaf padding are done at this level. The structure of the loop nest to walk the iteration space of trees and inputs is also decided at this level of the IR. TODO Should we mention that there is a scheduling language to decide this?

```
func Predict(float rows[batchSize]) {
  predictions = predictForest(rows)
  return rows
}
```

2.2 Mid Level IR

The Mid Level IR makes the loop structures and tree walks more explicit. Firstly, the order in which the iteration space of trees and inputs is walked is explicitly specified in the IR through loop nests. Also, operations such as `isLeaf`, `traverseTile`, `getLeafValue` are introduced so that the traversal of trees explicitly represented. The following listing shows the IR for inference using a model with four trees on an input batch with two rows. The listed IR walks all trees for one input row before moving to the next row. One important point to note here is that details such as the data structure used for the trees are not explicitly encoded in the IR. This allows us to reuse optimization and lowering passes on this level of the IR regardless of what the final in memory representation of the model is.

```
// Constant that represents the model being compiled
forest = ensemble(...)
for i = 0 to 2 step 1 {
  prediction = 0
  for t = 0 to 4 step 1 {
    tree = getTree(forest, t)
    node = getRoot(tree)
    while (isLeaf(tree, n)==false)  do {
      node = traverseTreeTile(tree, node, rows[i])
    }
    treePrediction = getLeafValue(tree, node)
    prediction = prediction + treePrediction
  }
  predictions[i] = prediction
}
```

The IR listed above is a simplification of the actual IR. The actual IR is strongly typed and in SSA form.

2.3 Low Level IR

The IR is finally lowered to a form where the in memory representation of the model is made explicit. Buffers to hold the model values are inserted into the generated code and all tree operations in the mid-level IR are lowered to explicitly reference these buffers. The semantics of all operations are made explicit. For example, `traverseTreeTile` is lowered into a series of operations to load thresholds, feature indices and features, compare the features with the thresholds and compute the next node to evaluate. This IR is then lowered directly to LLVM IR and JITted.

3. OPTIMIZATIONS

TODO This section needs a better name!

3.1 Notation

TODO Notation needs to be introduced in the background section We represent a decision tree by $T = (V, E, r)$ where V is the set of nodes, E the set of edges and $r \in V$ is the root. For each node $n \in V$, the following are given.

1. $threshold(n) \in \mathbb{R}$ which gives the threshold value for n.
2. $featureIndex(n) \in \mathbb{N}$ which gives the feature index for n.
3. $left(n) \in V$, the left child of n or $\emptyset$ if n is a leaf. If $left(n) \neq \emptyset$, then $(n, left(n)) \in E$.
4. $right(n) \in V$, the right child of n or $\emptyset$ if n is a leaf. If $right(n) \neq \emptyset$, then $(n, right(n)) \in E$.

We use $L \subset V$ to denote the set of leaves.

3.2 Tiling

Treebeard vectorizes tree walks by grouping nodes of a decision tree into ***tiles***. The nodes in a tile are evaluated concurrently using vector instructions. Once the nodes of the current tile are evaluated, a look up table is used to compute which child of the current tile to move to next. The listing below shows at a high level how a tiled tree is walked.

```
1  // A lookup table that determines the child index of
2  // the next tile given the tile shape and the outcome
3  // of the vector comparison on the current tile
4  int16_t LUT[NUM_TILE_SHAPES, pow(2, TileSize)]
5
6  ResultType Prediction_Function(...) {
7    // ...
8    Node n = getRoot(tree)
9    while (isLeaf(tree, n)==false) do {
10     thresholds = loadThresholds(tree, n)
11     featureIndices = loadFeatureIndices(tree, n)
12     // Gather required feature from the current row
13     features = rows[i][featureIndices]
14     // Vector comparison of features and thresholds
15     comparison = features < thresholds
16
17     // Pack bits in comparison vector into an integer
18     comparisonIndex = combineBitsIntoInt(comparison)
19
20     // Get child index of tile we need to move to
21     tileShape = loadTileShape(tree, n)
22     childIndex = LUT[tileShapeID, comparisonIndex]
23
24     // Move to the correct child of the current node
25     n = getChildNode(tree, n, childIndex)
26   }
27   ThresholdType prediction = getLeafValue(n)
28   // ...
29 }
```

To evaluate the current tile, the vector of thresholds is first loaded (`loadThresholds`). This vector contains the thresholds of all nodes in the tile. Then, the features required for comparison are gathered into a vector (lines 11 and 13). The feature vector is compared to the threshold vector and the child tile to move to is determined (lines 15 to 25). More details about tile shapes and the look up table are presented in subsequent sections.

3.3 Tiles and Tile Shapes

A ***tile*** is a collection of connected non-leaf nodes of a decision tree. The path connecting any pair of nodes in the tile must fully be contained within the tile. The tile size n_t is the number of nodes contained in a tile.

Informally, the ***tile shape*** is the shape of the region that encloses all nodes in a tile in a diagram of the decision tree. More formally, for a tile size n_t, each unique legal binary tree containing n_t nodes (nodes being indistinguishable) corresponds to a tile shape.

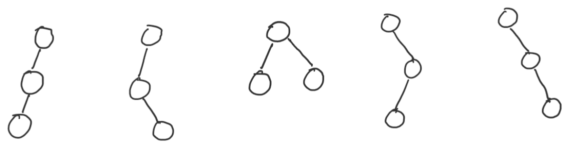
Figure 2: All possible tile shapes with a tile size $n_t = 3$

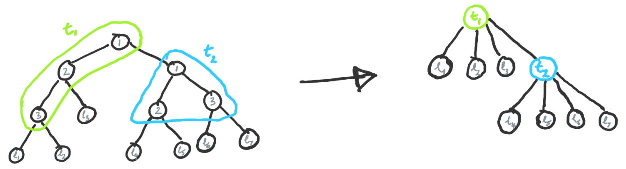

Figure 3: Example of a valid tree tiling with a tile size $n_t = 3$

Figure 2 enumerates all tile shapes with a tile size of 3. There are a total of 5 tile shapes with size 3. The number of tile shapes with a tile size n_t, denoted by $NTS(n_t)$ is given by the following equation.

$$NTS(n) = \sum_{k=0}^{n-1} NTS(k) \times NTS(n-k-1) \qquad (1)$$

where $NTS(0) = NTS(1) = 1$.

3.4 Valid Tiling of a Tree

A tiling $\mathscr{T}$ of the tree $T = (V, E, r)$ with tile size n_t is a partition $\{T_1, T_2, ..., T_m\}$ of the set V such that

1. $T_1 \cup T_2 ... \cup T_m = V$
2. $T_i \cap T_j = \emptyset$ for all $i, j \in [1, m]$ and $i \neq j$
3. $|T_i| \leq n_t$ for all $i \in [1, m]$
4. $\forall l \in L : l \in T_i \rightarrow v \notin T_i \;\; \forall v \in V \backslash \{l\}$
5. Tiles are **maximal**, i.e. if $|T_i| < n_t$, then there is no $v \in V \backslash \{T_i \cup L\}$ such that $(u, v) \in E$ for some $u \in T_i$.
6. Tiles are **connected**, i.e. for an $u, v \in T_i$, there is a (undirected) path connecting u and v fully contained in T_i.

TODO AP We need to talk about how tiling is specified in the compiler

3.5 Tiled Trees

A tiling transformation communicates the tiling to the Treebeard infrastructure by assigning a tile ID to each node in the decision tree. Using these tile IDs, Treebeard checks the validity of the tiling and then contructs a tree whose nodes are tiles. We call this tree the ***tree of tiles***. TODO We need a better name for this Figure 3 shows a valid tiling with tile size 3 and the tree of tiles constructed by Treebeard. Three nodes are grouped into each of the tiles t_1 and t_2 as shown. Each tile is collapsed into a single node in the tree of tiles. However, each leaf in the original tree becomes a leaf in the tree of tiles.

Treebeard maintains the following invariants.

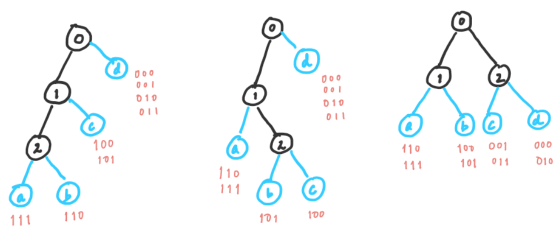

Figure 4: Example tile traversals with tile size $n_t = 3$

1. All tiles in a tree are the same size n_t. If the tiling produces any smaller tiles, these are padded by inserting dummy nodes to make them the required size.
2. Nodes within tiles are always ordered in level order and left to right within a level. The numbering of the nodes in the above diagram shows this node order.
3. Children of a node are numbered from left to right (regardless of level). For example, l_1 is the first child of t_1, l_2 is the second and so on.

3.6 Tile Shapes and Decision Tree Inference

Treebeard uses vector instructions to accelerate decision tree walks. Vector instructions are used to evaluate the predicates of all the nodes in a tile simultaneously. However, once the predicates of all the nodes in the tile are evaluated, computing the next tile to move to, given the outcome of the comparison depends on the tile shape of the current tile. To illustrate this problem, consider the case of the tiles of size 3 shown in figure 4. The diagram shows 3 of the 5 possible tile shapes for a tile size of 3. The nodes drawn in black are members of the tile t_1. The nodes in blue are the entry nodes of the children tiles of t_1. TODO Define entry nodes

To traverse a tile on an input row, first, the predicate of each node in the tile is computed. Subsequently, we need to determine which of the child tiles to move to next. Note that a true predicte (bit value 1) on a node implies a move to the left child and a false predicate (bit value 0) implies a move to the right child.

In the diagram, the bit strings (written in red) show which child we need to move to given the outcomes of the comparison (the bits represent the comparison outcomes of nodes and are in the order of the nodes in the tile – marked 0, 1 and 2 in the diagram, i.e., the MSB is the predicate outcome of node 0 and the LSB the predicate outcome of node 2). For example, for the first tile shape, if the predicate of all nodes are true (i.e. the comparison outcome is 111), the next node to evaluate is a. However, if the predicate of node 1 is false, then we need to move to d regardless of the outcomes of nodes 2 and 3. It is easy to see from the diagram that, depending on the tile shape, the same predicate outcomes can mean moving to different children. For example, for the outcome "011", the next tile is the 4th child (node d) for the first two tile shapes while it is the 3rd child for the other tile shape (node c).

3.7 Lookup Table

A lookup table (LUT) is used to solve the problem described in section 3.6, i.e. given the outcome of the comparisons on all nodes in a tile, determine the child tile we should evaluate next. The LUT is indexed by the tile shape and the comparison outcome. Formally, the LUT is a map.

$$LUT : (TileShape, < Boolean \times n_t >) \rightarrow [0, n_t] \subset \mathbb{N}$$

where n_t is the tile size, $< Boolean \times n_t >$ is a vector of n_t booleans. The value returned by the LUT is the index of the child of the current tile that should be evaluated next. For example, if we are evaluating the first tile t in figure 4, and the result of the comparison is 110, then $LUT(TileShape(t), 110) = 1$ since the tile we need to evaluate next is the tile with node b, which is the second child of the current tile.

In order to realize this LUT in generated code, Treebeard associates a non-negative integer ID with every unique tile shape of the given tile size. The result of the comparison, a vector of booleans, can be interpreted as a 64-bit integer. Therefore, the LUT can be implemented as a 2 dimensional array.

```
int16_t LUT[NTS(n_t), pow(2, n_t)]
```

Treebeard computes the values in the LUT statically as the tile size is a compile time constant.

3.8 In Memory Representation of Tiled Trees

Treebeard currently has two in memory representations for tiled trees - an array based representation and a sparse representation. Both representations use an array of structs to represent all tiles of the model.

3.8.1 *Array Based Representation*

Each tree in the model is represented as an array of tiles using the standard representation of trees as arrays. The root node is at index 0 and for a node at index n in the array, the index of its i^{th} child is given by $(n_t + 1)n + (i + 1)$ (every node in the tree of tiles has $n_t + 1$ children). A tile is represented by an object of the following struct.

```
struct Tile {
  // A vector of TileSize elements
  <ThresholdType x TileSize> thresholds;
  <FeatureIndexType x TileSize> featureIndices;
  // Integer that identifies the tile shape
  TileShapeIDType tileShapeID;
};
```

TODO AP Is this level of detail really needed? Also, the vector type notation needs to be introduced somewhere. Even though this representation is simple, the memory required for reasonable sized models is very large. The memory footprint is up to 20X that of the scalar representation. This memory bloat causes a performance degradation because the span of the L1 TLB is not sufficient to efficiently translate addresses for the whole model. TODO AP There are also some cache misses. Write this better. Storing leaves as full tiles (even though leaves just have to represent one value) and the empty space introduced due to the array based representation of trees that are not complete account for most of the increase. The sparse representation described next tries to address these issues.

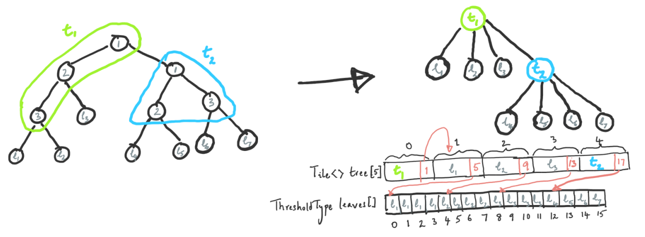

Figure 5: Sparse representation with tile size $n_t = 3$

3.8.2 *Sparse Representation*

The sparse representation tries to address the large memory footprint of the array based representation by doing the following.

- To eliminate the wasted space in the array representation, we add a child pointer to each tile. This points to the first child of the tile. All children of a tile are stored contiguously.
- Leaves are stored in a separate array. We found that, across all our benchmarks, a large fraction of leaves are such that all their siblings are also leaves. Such leaves are directly moved into the leaves array. For leaves for which this property does not hold, an additional hop is added by making the leaf tile a comparison tile and all its children are made leaves with the same value as the original leaf.

Figure 5 shows some of the details of the sparse representation. The tree on the left of the diagram is the actual decision tree with the nodes grouped into tiles t_1 and t_2. The tree on the right is the tree of tiles. The arrays depicted below show how the tree is represented in memory. The first array (`tree`) is an array of tiles and has 5 elements. Each element of the array represents a single tile and has the thresholds of the nodes, the feature indices, a tile shape ID and a pointer to the first child (shown explicitly in red).

As a specific example, consider the tile t_1. The tile has four children – l_1, l_2, l_3 and t_2 in that order (left to right). These tiles are stored contiguously in the `tree` array and a pointer to the first of these, l_1 is stored in the tile t_1 (the index 1 is stored in the tile t_1 as shown).

Now consider the tile t_2. Since all children of the tile t_2 are leaves, they are all moved into the `leaves` array. To store a pointer into the `leaves` array, we add `len(tree)` to the element index in the `leaves` array. The tile t_2's child is the element at index 12 of the `leaves` array. Therefore, the index $12 + 5 = 17$ is stored in the tile t_2. (Any index i that is greater than the length of the `tree` array is regarded as an index into the `leaves` array. The index into the `leaves` array is $i -$ `len(tree)`.)

The other aspect of the representation is that an extra hop is added for the leaves l_1, l_2 and l_3 in order to simplify code generation. This enforces the invariant that all leaves are stored in the leaves array and simplifies checking whether we've reached a leaf. Therefore, 4 new leaves are added as children for each of the original leaves l_1, l_2 and l_3. Each of these 12 newly added leaves has the same value as its parent. These are the first 12 elements of the `leaves` array.

Even though we currently have implementations of the two representations detailed in sections 3.8.1 and 3.8.2, support for other representations is not hard to add. All optimizing passes that work on the high level and mid level IR will continue to work as is. Programmers need only provide new lowering passes for a few operations in the low level IR.

3.9 Uniform Tiling

The first tree tiling algorithm we implement is uniform tiling. The basic idea of uniform tiling is to construct tiles with a tile size n_t so that we evaluate the least possible number of nodes speculatively. Intuitively, this translates to minimizing the height of the constructed tile. We minimize the height of the tile by performing a level order traversal starting at the root node of the tile. TODO AP Should we say this is optimal when probabilities are equal? Problem is we haven't said what optimal is. This traversal collects at most n_t nodes and does not include any leaves in the tile so that the constraints listed in section 3.4 are respected. Algorithm 1 lists the algorithm for uniform tiling. Tiling starts at the root and constructs a tile *Tile* by performing a level order traversal. Once the current tile is constructed, the tiling procedure is recursively performed on all nodes that are destinations for edges going out of the constructed tile. The set of tiles constructed by algorithm 1 constitutes a uniform tiling of the input tree.

Algorithm 1 Uniform tree tiling

```
procedure TILETREE($T = (V, E, r)$, $n_t$)
    if $r \in L$ then
        return $\{r\}$
    end if
    //Level order traversal to collect $n_t$ or fewer nodes.
    //Leaves are not included in the constructed tile.
    $Tile \leftarrow LevelOrderTraveral(r, n_t)$
    $Tiles = \{Tile\}$
    for $(u, v) \in Out(Tile)$ do
        $Tiles \leftarrow Tiles \cup TileTree(T_v, n_t)$
    end for
    return $Tiles$
end procedure
```

3.9.1 Further Opimization and Code Generation

We found that most leaf tiles for a given tree are at the same depth when uniform tiling is used. Furthermore, we see that deeper leaves are more likely to be reached. Based on these observations, we pad the tree of tiles generated with uniform tiling so that all leaves are at the same depth. This transformation is performed on the high level IR after uniform tiling. Once the trees have been padded to make all leaves equal depth, the tree walks are fully unrolled to evaluate a fixed number of tiles and all leaf checks are elided.

One other complication the code generator needs to handle is the fact that trees potentially have different depths. In order to handle this, Treebeard sorts the trees by their depth. This ensures that all trees with equal depth are grouped together. Once this is done, the loop over the trees is fissed so that each of the resulting loops only walks trees of a single depth. Consider for example a forest with 4 trees T_1, T_2, T_3, and T_4 in that order. Further, assume that T_1 and T_4 have depth 2 while T_2 and T_3 have depth 3. First, Treebeard reorders the trees to be in the order T_1, T_4, T_2, T_3. Then, the loop over the trees is fissed as shown in the following listing.

```
forest = ensemble(...)
for i = 0 to batchSize step 1 {
  prediction = 0
  for t = 0 to 2 step 1 {
    tree = getTree(forest, t)
    node = traverseTreeTile(tree, node, rows[i])
    treePrediction = getLeafValue(tree, node)
    prediction = prediction + treePrediction
  }
  for t = 2 to 4 step 1 {
    tree = getTree(forest, t)
    node = traverseTreeTile(tree, node, rows[i])
    node = traverseTreeTile(tree, node, rows[i])
    treePrediction = getLeafValue(tree, node)
    prediction = prediction + treePrediction
  }
  predictions[i] = prediction
}
```

TODO AP: This listing is unnecesarily long. Can we maybe leave out the loop bodies and say something like "depth 2 walk"?